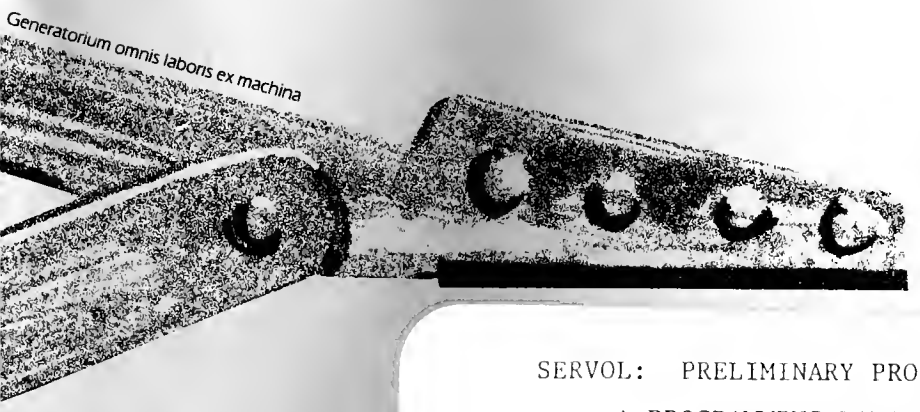
Robotics Research Technical Report



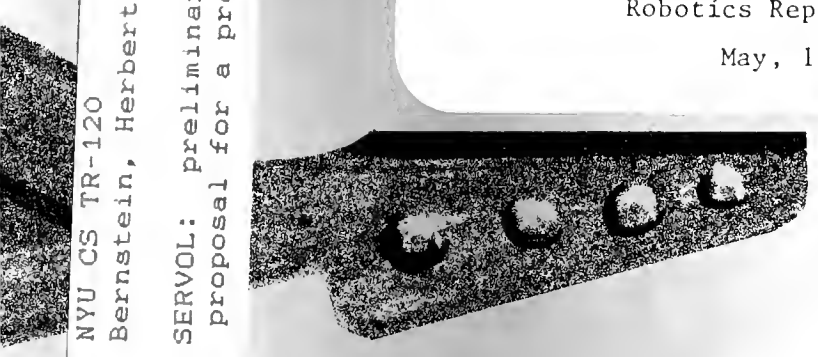
SERVOL: PRELIMINARY PROPOSAL FOR
A PROGRAMMING LANGUAGE
FOR REAL-TIME SERVO CONTROL

by

H.J. Bernstein, P.G. Lowney
and J.T. Schwartz

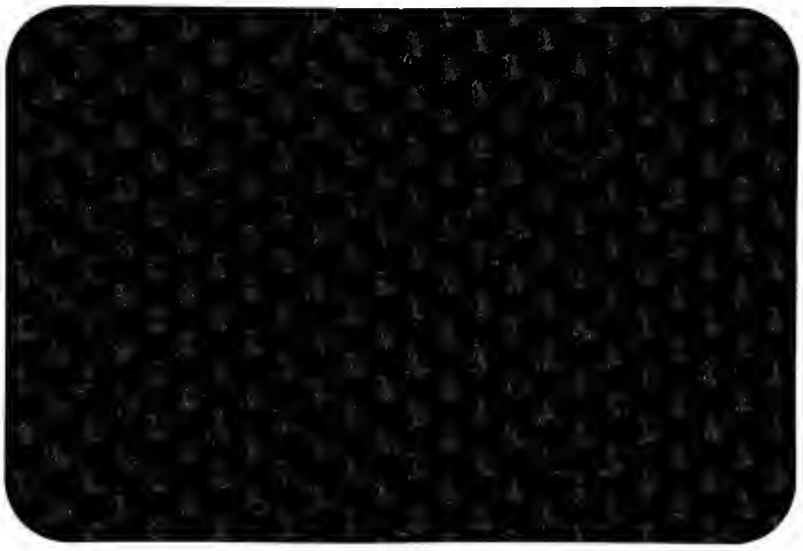
Technical Report No. 120
Robotics Report No. 26

May, 1984



New York University
Courant Institute of Mathematical Sciences

Computer Science Division
251 Mercer Street New York, N.Y. 10012



SERVOL: PRELIMINARY PROPOSAL FOR A PROGRAMMING LANGUAGE FOR REAL-TIME SERVO CONTROL

by

H.J. Bernstein, P.G. Lowney
and J.T. Schwartz

Technical Report No. 120
Robotics Report No. 26

May, 1984

Work on this paper has been supported in part by Office of Naval Research Grant N00014-82-K-0381, and by grants from the Digital Equipment Corporation, the Sloan Foundation, the System Development Foundation, and the IBM Corporation.

SERVOL
Preliminary Proposal for a Programming Language for Real-Time Servo Control

H.J. Bernstein
P.G. Lowney
J.T. Schwartz
Courant Institute of Mathematical Sciences
New York University

1. Introduction

This note will describe a language called SERVOL, designed for writing the innermost levels of real-time control for robotics and other situations in which sensors need to be read, and effectors controlled, on a continuing, high-frequency basis. SERVOL is non-procedural, and permits the programmer to specify succintly the relationships which must hold between senor input and control output. Because of the non-procedural design of SERVOL, the compiler can guarantee sensors will be read and control ports written within specified time limits.

SERVOL's relatively limited semantics make it suitable for implementation on a microprocessor. It is intended to run under the control of a host computer, which we will call HOST, which supplies parameters to the real-time SERVOL code, polls continuously changing values calculated by SERVOL, and to which SERVOL can communicate interrupts. Of course, the HOST computer need not be physically distinct from the microcomputer on which SERVOL executes. All that is essential is that SERVOL should be guaranteed short but sufficient periods of execution which recur regularly. In this sense, the SERVOL interpreter can be regarded as a somewhat exceptional, extremely privileged process within the HOST computer's operating system. Note however that the interface between SERVOL and this operating system is very narrow, i.e. SERVOL relates to the remainder of the system as a well-isolated module. Being nonprocedural, SERVOL relies on the parameter-setting activities of its HOST for all required procedural support.

The SERVOL system includes program development aids to meet to the special requirements of real-time control software. It provides facilities for limiting the range of crucial control parameters to prevent buggy control code from damaging external equipment. It also provides various output utilities, useful for debugging, which allow various displays of time-varying quantities to be set up easily. These output utilities can run on the HOST if the microprocesor executing SERVOL is too busy to handle them.

2. The SERVOL Language

Declarations

SERVOL provides three classes of variables, which are as follows.

CONTROL variables enable communication with a device being controlled. Every such variable is either a 'read only' control variable or a 'write only' control variable. Read only CONTROL variables, declared with the attribute READ, are quantities read from external sensors, and are available to SERVOL in input ports which SERVOL cannot modify. Examples are: the current position of a joystick; the force read from a tactile sensor. Write only CONTROL variables, declared with the attribute WRITE, are quantities which SERVOL writes to control ports, (which may be connected to D/A converters) to supply values to external devices. These ports are under the exclusive control of SERVOL. Examples are: the voltage supplied to a motor, the current supplied to an electromagnet.

PARAMETER variables enable communication with the HOST computer. Every such variable is either 'read only' or 'write only'. Read only PARAMETER variables, declared with the attribute READ, are quantities the HOST computer transmits to SERVOL. Examples are: the position target for a moving robot arm, position limits, thresholds at which interrupts are to be generated, etc. Write only PARAMETER variables, declared with the attribute WRITE, are quantities which SERVOL updates continuously and makes available to the HOST computer, which can poll them when required, use them to update graphic displays, etc.

INTERNAL variables are named intermediate quantities local to SERVOL, and are used to facilitate the computation of CONTROL WRITE and PARAMETER WRITE outputs from CONTROL READ and PARAMETER READ inputs. INTERNAL variables can be initialized to constant values.

SERVOL also allows symbolic names to be assigned to frequently used CONSTANTS. CONSTANTS are local to SERVOL. TRUE and FALSE are preclared constants, bound to the values one and zero respectively.

In addition, SERVOL variables can be specified to carry information about their past history, in a manner to be described below.

These various classes of variables are declared as follows in a SERVOL program:

```
(read-only control)              CONTROL   varname READ  PORT sysname
(write-only control)             CONTROL   varname WRITE PORT sysname
(read-only parameter)            PARAMETER varname READ  HOST sysname
(write-only parameter)           PARAMETER varname WRITE HOST sysname
(internal)                       INTERNAL  varname
(internal, with initialization) INTERNAL  varname := const
(constant)                       CONSTANT  varname := const
```

where varname is a lexically well-formed SERVOL variable name, sysname is a system name known to the SERVOL compiler, and const is a constant value.

Examples are:

```
CONTROL
   altitude   READ   PORT altitudeSensor,
   motorSpeed WRITE  PORT motorControl;
```

This statement declares two CONTROL variables: altitude, a read only CONTROL variable which reads the input port for an altitude sensor, and motorSpeed, a write only CONTROL variable which writes the output port for a motor controller. Both altitude and motorSpeed are lexically well-formed SERVOL variable names, whereas altitudeSensor and motorControl are names for control ports known to the SERVOL compiler. (Note that a version of the SERVOL compiler is targeted to each microprocessor board on which SERVOL is to execute, and each version of the compiler will allow some preassigned list of control port names.)

```
PARAMETER
   arriving READ  HOST alpha,
   feedBack WRITE HOST beta;
```

This statment declares two PARAMETER variables: arriving, a read only PARAMETER variable which reads the value of the HOST variable alpha, and, feedBack, a write only PARAMETER which writes the host variable beta. Again, arriving and feedBack are lexically well-formed SERVOL variable names, whereas alpha and beta are names for HOST variables known to the SERVOL system. The actual transmission of parameter values from the HOST to SERVOL is system dependent, and may involve the HOST computer directly accessing the memory of the SERVOL microprocessor, or may utilize some more complicated parameter-setting protocol.

```
INTERNAL
        initFlag  := TRUE,
        counter   := 0;
```

This statment declares two INTERNAL variables: initFlag and counter, and initializes them to TRUE and 0 respectively.

```
CONSTANT
        fieldWidth := 3;
```

declares fieldWidth to be a constant bound to 3.

The value of a variable is a bit string of machine word size. An alternative size can be specified in the declaration by the key word SIZE. For example

```
INTERNAL
        small SIZE 7;
```

declares small to be a bitstring of length 7.

Assignments and Expressions

The basic SERVOL statement is the assignment, whose simplest form is

```
v := expn;
```

in which 'v' is a legal variable name and 'expn' is a wellformed expression. Such an expression can involve the arithmetic, relational, and boolean operators, and conditional expressions.

The SERVOL data type is a bit string. However, the bit string may be interpreted by the various operators as a boolean value (0 is false, every thing else is true), a two's complement integer, or a bit string. The bits in a word are numbered from one to the length of the string, and one indexes the least significant bit. Subfields in a word can be extracted and assigned: x(i..j) selects a field starting in bit position i and ending at bit position j from the variable x. i and j must be constants for efficiency, and to permit the enforcement of the SERVOL single assignment rule (see below).

The general form of the SERVOL assignment statement is

```
lhs := expn;
```

in which 'lhs' is a legal variable name or field extraction and 'expn' is a wellformed expression. Such an expression can use the arithmetic operators +, -, *, /, the relational operators <, <=, >, >=, =, /=, and the boolean operators AND, OR, and NOT. Standard precedence is used. Conditional if-then-else expressions are also allowed

SERVOL assignment statements satisfy a single-assignment rule: no bit in a variable may be the target of more than one assignment statement. More than one field in a variable can be the target of an assignment if the fields do not overlap. An assignment to a variable x can be viewed as an assignment to x(1..L), where L is the length of x. Thus an assignment to x prevents any subsequent assignment to a sub-field of x.

For syntactic elegance, we allow assignments of the form

```
(1) v := If c1 THEN e1 ELSEIF c2 THEN e2...ELSE en END IF;
```

to be written as a contiguous block in the form

```
(2)  v := e1 WHEN c1,
     v := e2 WHEN c2,
        ...
     v := e(n-1) WHEN c(n-1),
     v := en OTHERWISE;
```

The semantic significance of a SERVOL assignment statement is that it holds true 'continuously'. That is, the microprocessor executing SERVOL executes all assignments repeatedly, as often as it

can (e.g. once each millisecond). Thus all assignments are 'always' true, at roughly a millisecond level of accuracy. For examply, if JOYSTICK_POSITION is a CONTROL READ variable, SCALE is a PARAMETER, and VOLTAGE is a CONTROL WRITE variable, then by writing

```
VOLTAGE := SCALE * JOYSTICK_POSITION;
```

we cause the output VOLTAGE to track the measured JOYSTICK_POSITION very closely.

These 'continuous' assignments are effected by repeatedly evaluating the assignment statements in a well-defined intrepretation loop. More precisely, variable revisions take place at the following moments during the SERVOL interpreter loop:

(i) All CONTROL and PARAMETER variables with the READ attribute are read, and their values fixed for this cycle.

(ii) All expressions are evaluated;

(iii) All variables are assigned their new values;

(iv) The next cycle of SERVOL interpretation begins.

Certain PARAMETER WRITE variables may have interrupt-like effects. That is, a characteristic interrupt will be transmitted to the HOST computer whenever a non-zero value is assigned to such a variable.

Because expressions are evaluated before assignments are made in the interpretation loop, circularities between SERVOL assignments are permitted. For example, the following (rather artificial) program models a flip-flop, with a clock rate corresponding to the cycle time of the SERVOL interpretation loop:

```
PARAMETER
   set   READ   HOST flipFlopSet,
   reset READ   HOST flipFlopReset,
   q0    WRITE  HOST flipFlopState,
   q1    WRITE  HOST flipFlopStateBar;

q0 := 1 WHEN set,
q0 := 0 WHEN reset,
q0 := NOT q1 OTHERWISE;

q1 := 0 WHEN SET,
q1 := 1 WHEN RESET.
q1 := NOT q0 OTHERWISE;
```

History Mechanism

It will sometimes be useful to keep the recent past history of a SERVOL variable available. This can be done by attaching one of two suffixes to its declaration. These suffixes are

(3) HISTORY PAST n INTERVALS t

and

(4) HISTORY PAST n

where t is a positive integer representing periods of time expressed in milliseconds, and n is a positive integer indicating the number of old values to maintain. Examples are:

```
(5) CONTROL
       pressure    READ  PORT gauge1  HISTORY PAST 5 INTERVALS 200,
       temperature READ  PORT qauge2  HISTORY PAST 10;
```

When attached to the declaration of a variable, these clauses create additional 'generations' of the variable, which record the history of the variable, either at a succession of past instants (Case (3) above) or upon successive assignments (Case (4)). In either of these cases, the j-th past value of the variable can be retrieved by writing v(j). For example, the first statement of (5) causes 5 past values of the gauge1 input to be maintained, as pressure(1). .pressure(5) (from most recent past to most distant past). (Note pressure(1) is equivalent to pressure.) This makes it possible to estimate the rate of pressure change by writing

```
rate := (pressure(1) - pressure(2))/200;
```

If desired, a more sophisticated differentiation formula can be used to estimate this derivative.

The historic values of a variable v whose declaration carries the clause (3) will be updated every t milliseconds. Otherwise, they are updated when a value is assigned to the variable. (Note because of conditional assignments, a variable may not be assigned a value on every iteration of the interpretation loop.)

On startup, the compiler will assign the intial value of the variable to all of its historic values. The initial value is the first reading of values for CONTROL READ and PARAMETER READ variable, the first assignment for all CONTROL WRITE and PARAMETER WRITE variables, and the intilization or first assignment for all INTERNAL variables.

The time at which the last update of a variable v occurred is available through the operator

```
UPDATE v.
```

The value of the update time associated with a variable is itself

updated in the step (iii) of the interpretation loop.

The current time value is available through the special variable TIME; it is assigned a value at step (i) of the interpretation loop.

Additional Constructs

To allow convenient grouping of sets of SERVOL statements whose effects must be switched on and off together, we allow the following IF expression form:

```
IF condition_1 THEN
        block_1 of statements
ELSEIF condition_2 THEN
        block_2 of statements
   ...
ELSEIF condition_n THEN
        block_n of statements
ELSE
        block of statements
END IF;
```

These IF statements can be nested, and the ELSE portion omitted.

The SERVOL 'single assignment' rule applies in the following way to such a IF statement: separate assignments to a given field f of a variable v can occur in the various blocks-of-statements (since only one of these assignments will be executed during a given cycle of the SERVOL interpreter.) If any such assignment occurs within a IF, then the whole IF is considered to involve a single assignment to f. For nested IFs, this rule is applied recursively.

As a convenience, we allow expressions to be written as functions in a style resembling that used in FORTRAN statement functions, i.e.

```
FUNCTION function-name(p1,...,pn)  expn;
```

when the expression on the right must depend only on the parameters appearing on the left, and direct or indirect recursion is not permitted. The SERVOL compiler will expand functions in-line, and they should be viewed by the programmer as a restricted macro facility.

Sequential Evaluation

It is occaisonally convenient in SERVOL programs to use assignments which are performed sequentially rather than in parallel. We do this with the keyword NEXT. An occurrence of NEXT in the program forces all assignments preceding it in the program text to be performed. The new values of the assigned variables can be accessed in the evaluation of the expressions following the NEXT.

For example, in the following SERVOL program the final value of x will be -1.

```
INTERNAL
        x := 5,
        flag := TRUE;

flag := x > 0;
x := x - 1 WHEN flag;
```

However, with the introduction of a NEXT, the final value of x will be 0:

```
INTERNAL
        x := 5,
        flag := TRUE;

 flag := x > 0;
NEXT;
 x := x - 1 WHEN flag;
```

A NEXT statement cannot occur inside an IF statement, and it cannot be qualified with a WHEN.

The occurence of more than one NEXT in a program divides it into blocks which are evaluated in sequential order. If there are n-1 NEXT's in a program, then there are n blocks. Let block1 be the block of statements before the first NEXT, block2 be the block of statments between the first and second NEXT, and blockn be the block of statements after the last NEXT. Then the SERVOL interpretation loop is now:

(1) All CONTROL and PARAMETER variables with the READ attribute are read, and their values fixed for this cycle.

(2) All expressions in block1 are evaluated.

(3) All assignments in block1 are performed.

(4) All expressions in block2 are evaluated.

(5) All assignments in block2 are performed.

.....

(2n) All expressions in blockn are evaluated.

(2n+1) All assignments in blockn are performed.

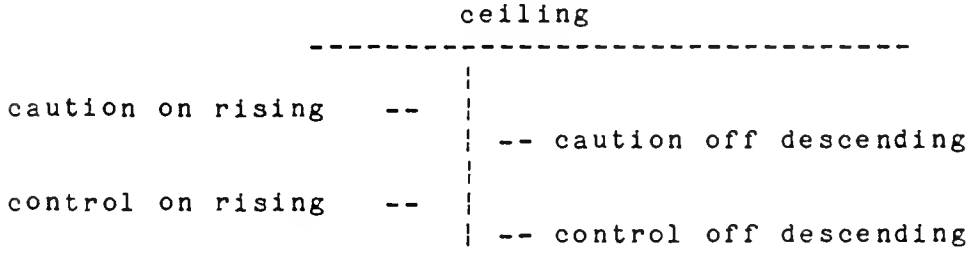
(2n+2) The next cycle of SERVOL interpretation begins.

Some Programming Examples

Consider the problem of raising a robot arm to touch the ceiling. We assume we have a motor controlling the arm and an altitude sensor which indicates its height from the floor. When the arm is well away

from the ceiling, we want to move it a maximum speed. As we approach the surface we must slow the arm down, and as we are about to touch it we must run at the minimum speed. To leave the surface we must gradually speed up the arm. We don't worry here about stopping the arm completely as it touches the ceiling.

We introduce two thresholds to determine how to control the arm speed. When above the control threshold, the speed of the arm must be controlled (i.e., run at less than maximum speed), and above the caution threshold the arm must move at a minimum speed. The altitude sensor is biased by the motion of the arm, and will overestimate the true altitude when moving up and underestimate it when moving down. Thus thus when approaching the ceiling we use higher thresholds than when we are leaving. The situation is as follows:



On approaching the ceiling, we must start slowing the motor when the arm crosses the 'control on rising' threshold, and we must set the motor to the minimum speed when crossing the 'caution on rising' threshold. When descending, we continue at minimum speed until the arm crosses the 'caution off descending' threshold, and slowly raise the speed until the arm reaches the 'control off descending' threshold, where we no longer need be concerned about the ceiling.

The SERVOL program to control the arm's motor is as follows. Note we control only the speed of the motor, not the direction of the motion.

```
CONTROL
        altitude    READ          PORT altitudeSensor,
        motorSpeed  WRITE         PORT speedControl;

PARAMETER
        arriving    READ          HOST a1,
        interval    READ          HOST a2,
        controlOn    READ         HOST a3,    $ control on rising
        controlOff  READ          HOST a4,    $ control off descending
        cautionOn   READ          HOST a5,    $ caution on rising
        cautionOff     READ       HOST a6,    $ caution off descending
        minimumSpeed  READ        HOST a7,
        maximumSpeed  READ        HOST a8;

INTERNAL
        controlRequired   := FALSE,
        cautionRequired   := FALSE;


 controlRequired := true WHEN altitude > controlOn,
 controlRequired := false WHEN altitude < controlOff,
 controlRequired := controlRequired OTHERWISE;

 cautionRequired := true WHEN altitude > cautionOn.
 cautionRequired := false WHEN altitude < cautionOff,
 cautionRequired := cautionRequired OTHERWISE;

NEXT;

 IF (TIME - UPDATE motorSpeed) > interval THEN
        motorSpeed := minimumSpeed WHEN cautionRequired,
        motorSpeed := motorSpeed - 1 WHEN controlRequired
                                     AND arriving
                                     AND motorSpeed > minimumSpeed,

        motorSpeed := motorSpeed + 1 WHEN controlRequired
                                     AND NOT arriving
                                     AND motorSpeed < maximumSpeed,

        motorSpeed := maximumSpeed OTHERWISE;
END IF;
```

Next consider the problem of system reinitializations. These can be triggered by the HOST computer, which has only to assign a value n denoting the desired initialization to an appropriate PARAMETER quantity q, and then to toggle a second PARAMETER quantity doInit between zero and 1. The appropriate SERVOL structure is

```
PARAMETER
        q       READ    HOST a1,
        doInit  READ    HOST a2;

INTERNAL
        reInit,
        oldInit := FALSE;

  reInit := NOT (oldInit = doInit);
  oldInit := doInit;

NEXT;

   IF reInit AND Q=1
      (Initialization sequence 1)
   ELSEIF reInit AND Q = 2
      (Initialization sequence 2)
    ...
   ELSE
      (standard actions when no re-initialization)
   END;
```

Suppose next that we wish to update some quantity u periodically, e.g. every 10 milliseconds, starting from a given moment at which u is initialized. This can be done using the UPDATE function:

```
PARAMETER
        interval  READ  HOST a1;

INTERNAL
        u;

   u := (appropriate expression) WHEN (TIME - UPDATE.u) > interval;
```

As a third example, consider the servol control of a positioner, i.e. we apply a force to a positioner, measure the error in position and use a function of that error to generate a revised force. (See George J. Thaler, Robert G. Brown, "Servomechanism Analysis", McGraw-Hill, New York, 1953, 414 pp.)

In the simplest case, the function of the error is linear homogeneous. We are given a servo gain by which to multiply the positional error, to produce a force. The controlled output is force. The sensed input is position. The host provides a target position and a servo gain.

```
CONTROL
        force     WRITE          PORT forceRegister,
        position  READ           PORT positionRegister;

PARAMETER
        targetPosition READ      HOST a1,
        servoGain READ           HOST a2;

force := servoGain * (position - targetPosition);
```

This simple model will serve to maintain position in the absence of load, provided we know a suitable servo gain. If the positioner is loaded, we may have difficulty in achieving zero error at low gains, yet at high gains we face the problem of highly oscillatory behavior.

In order to improve positioning responsiveness and cope with loads, it usually is necessary to add corrections for the rate of change of error and for the accumulated error, each with its own gain. The accumulated error is just a summation, though in practice a damped summation might be used to reduce the effect of past large excursions. The rate is a difference between current and past errors.

```
CONTROL
        force WRITE                      PORT forceRegister,
        position READ                    PORT positionRegister;

PARAMETER
        targetPosition READ              HOST a1,
        positionServoGain READ           HOST a2,
        velocityServoGain READ           HOST a3,
        accumulatedErrorGain READ        HOST a4;

INTERNAL
         error := 0 HISTORY PAST 2,
         accumulatedError := 0;

error := position - targetPosition;
accumulatedError := accumulatedError + error

force := positionServoGain * error
           + velocityServoGain * (error(1)-error(2))
           + accumulatedErrorGain * accumulatedError;
```

As a final example, suppose we wish to have a robot which is moving a constant speed steer along a line using a photosensor. Suppose the sensor is an array of 16 photocells which are reported as bits in a word. A cell over the line will cause a bit of 1. Otherwise we expect zero. We divide the sensor into zones. In the middle we have two zones to start us steering. Beyond that we have two zones to force a sharper turn. The outer

sensors will cause a panic stop because we are out of control. The host must provide us with two steering increments and a steering limit. The limit will be taken as somewhat soft, just to avoid unnecessary complications in the example.

```
CONTROL
        direction WRITE         PORT steeringRegister,
        panicStop WRITE         PORT panicStopRegister,
        sensor    READ          PORT sensorRegister;

PARAMETER
        innerRate READ          HOST a1,
        outerRate READ          HOST a2,
        limit     READ          HOST a3;

INTERNAL
        localDirection:=0;

FUNCTION guardZone()            sensor(0..1,14..15);

FUNCTION totalField()           sensor(0. 15);

FUNCTION leftOuterZone()        sensor(2..4);

FUNCTION leftInnerZone()        sensor(5..7);

FUNCTION rightInnerZone()       sensor(8..10);

FUNCTION rightOuterZone()       sensor(11. 13);


panicStop :=1 WHEN guardZone() /= 0 OR totalField() = 0,
panicStop := 0 OTHERWISE;

localDirection := localDirection + innerRate
          WHEN rightInnerZone() = 0 AND leftInnerZone() /= 0,

localDirection := localDirection - innerRate
          WHEN leftInnerZone() = 0 AND rightInnerZone() /= 0,

localDirection := localDirection + outerRate
          WHEN leftOuterZone() /= 0,

localDirection := localDirection - outerRate
          WHEN rightOuterZone() /= 0,

localDirection := limit WHEN localDirection > limit,

localDirection := -limit WHEN localDirection < limit;


direction := localDirection;
```

Note that it is possible the several of the conditions for changing the local direction may be satisfied, but that only one of them will have effect. If we cared about which one, we would nest the conditionals instead of leaving them at the outer level. As it is, the semantics of SERVOL prescribe that the first condition found to be valid will trigger an assignment to localDirection and the others will be bypassed.

3. The SERVOL Environment

Output Utilities

SERVOL provides various output utilities, useful for debugging, which allow convenient displays of time varying quantities to be set up easily. These output utilities can run on the HOST computer. The set of utilities provided in any SERVOL implementation will depend on the sophistication of the display device attached to the HOST. In the following purely illustrative discussion we describe display primitives, some of which are only suitable for a bit-mapped color display screen.

The simplest display primitive is

```
DISPLAY v1,v2, . ,vn IN w;
```

Here v2,v2....vn are variable names, and w is a pair of numbers designating a 'window' on the display screen, i.e. a rectangular area. This command causes a dynamic display having the form

```
v1 = *****
v2 = *****
  ...
vn = *****
```

to appear in the 'window' area defined by w. The fields designated by ***** will at all times contain the values of the dynamically varying quantities v1,...,vn. Thus, for example, to create an electronic clock display of the kind sometimes seen in store windows, one has only to issue the command

```
DISPLAY TIME;
```

The variables in the DISPLAY list can be grouped together in parentheses, and then all the variables in a group will be displayed on the same line.

The command

```
CLOSE w;
```

where as before w designates a window, will terminate all displays overlapping this window, and blank the areas occupied by these displays.

The command

```
GRAPH i1,i2, ..,in PAST t IN w;
```

produces a different sort of dynamic display, resembling what would be

seen after opening a window of fixed size on a multi-pen laboratory paper tape recorder. As before, w defines the screen 'window' in which the specified display will appear. In this GRAPH command each of the items ij should be a parenthesized pair consisting of a variable name and a constant numerical 'line quality designator'. For a color screen, this designator defines the color in which the graphical line representing the values of the associated variable vj will appear, and may also define some other distinguishing quality of this line, e.g. its thickness, dot-dash pattern, etc. (Of course, on a screen without color, only these line qualities will be available.)

The parameter t in the GRAPH statment must be a constant designating the number of milliseconds of history of the variables v1,....vn to be shown in the dynamically varying graph. The graph produced is scaled to the size of the window in which it will appear and to this number of milliseconds. The graphs produced slide continuously from right to left. For example, the code

```
X = TIME MOD 2000;
```

followed by the command

```
GRAPH X PAST T IN w;
```

will produce endless an step function in the window specified by w.

Two additional SERVOL output commands are

```
SHOW i1,i2,....in IN w;
```

and

```
SHOW i1,i2, ..,in PAST t IN w;
```

In each of these commands, w designates some screen window, and the ij are quadruples (x, y, c, r) of four quantities. The display produced by the first form of SHOW statement appears as a collection of dots which move continuously in the specified window w. The position of each dot is determined by the parameters x,y; its color and intensity are determined by the corresponding parameter c, and its radius by the parameter r. Overall, the display produced by this SHOW statement appears as a collection of moving dots, which to convey additional information also change continuously in tint, intensity, and size.

The second form of the SHOW statement retains past positions of the collection of moving dots, thus converting them from circles to curved 'trails' which may convey a better sense of variation through time. Past positions back to a time t seconds before the present are retained. The intensity with which past positions are displayed falls off linearly, from full intensity for the present position of each dot, to zero intensity for its position t seconds previously.

'Guarding' the Values of Variables; Obligatory Guards:

SERVOL provides facilities for limiting the range of crucial control parameters to prevent buggy control code from damaging external equipment. These controls can be imposed in a manner which makes it impossible for unauthorized users to circumvent them. This gives essential protection during the development of new control algorithms in an experimental setting such as a student laboratory. In a manner that we will now explain. control is guaranteed by exploiting SERVOL's 'single assignment' rule.

Specifically, suppose that port1 denotes some output port whose values we wish to limit in a manner impossible to circumvent. This register will be made accessible to SERVOL through some declaration, say

```
(6)          CONTROL accessP1 WRITE PORT port1;
```

A given output port is allowed to appear in only one such declaration. This rule makes accessP1 the only name under which the port1 can be accessed in a program within which (6) appears. Then, to keep the values assigned to accessP1 from being unreasonable, one simply does not make it available directly to an 'unauthorized' user. Instead, such a user is compelled to reach accessP1 indirectly, through another variable, which for definiteness' sake we will call userAccessP1. Values assigned to userAccessP1 are subsequently checked by statements like the following:

```
(7)  outOfRange := (userAccessP1 < lowLimit) OR
                                (userAccessP1 > highLimit);

   NEXT;

     accessP1 := userAccessP1 WHEN NOT outOfRange;
     interruptCause := 3 WHEN outOfRange;
     interrupt := TRUE WHEN outOfRange;
```

These sample lines of code assume that interruptCause is a PARAMETER WRITE variable, that the HOST compute understands the significance of the numerical value 3, and that INTERRUPT is a PARAMTER WRITE variable which transmits a HOST interrupt whenever a nonzero value is assigned to it.

To enforce the limitations which the code (7) embodies, one has only to compel users to include these lines with their own SERVOL source code. Note that the single assignment rule then makes any other assignment to the underlying register accessP1 illegal.

Inclusion of 'protective' source like (7) is guaranteed as follows. At the start of each compilation run, SERVOL reads a system file which has the following format:

```
     PASSWORD w1:
             (lines of SERVOL source)
     PASSWORD w2:
             (lines of SERVOL source)
       ...
```

```
PASSWORD wn:
        (lines of SERVOL source)
ELSE
        (lines of SERVOL source).
```

Each wj is some encrypted quantity, 64 bits long. If no password, or a password with an encryption distinct from every wj, is supplied, the compiler prefixes the lines of code following the keyword 'ELSE' to the user-supplied source code. If a password mapping to wj is supplied, then the lines of code following the corresponding PASSWORD wj are prefixed to the user-supplied source. This option allows checks to be disabled by privileged users who may need to perform unusual (and perhaps dangerous) experiments. To supply a password, one invokes the SERVOL compiler with an optional parameter, and the compiler then interactively requests a password.